AdaCore TECH PAPER

The Work of Proof in SPARK

The Work of Proof in SPARK

Author: Claire Dross, AdaCore, 75009 Paris, France; email: dross@adacore.com

Abstract

Since Ada targets safety-critical programs, many features of the language introduce safety nets in the form of language-mandated checks. Even if compile-time verification is preferred to runtime verification whenever possible, many of these checks are still done dynamically, an exception being raised in case of violation.

The addition of contracts in Ada 2012 follows a similar trend, as a violation causes an exception to be raised when the code is compiled with assertions enabled. The SPARK tool aims at statically verifying that language-mandated checks and the user-written contracts can never fail at runtime. In this article, we give insights on how the tool works in practice and what are the most important challenges as of today.

Keywords: SPARK, proof of program, deductive verification.

1 Introduction

Since Ada targets safety-critical programs, many features of the language introduce safety nets in the form of language-mandated checks. For example, a check is made on every signed integer computation to make sure that it will not exceed the bounds of the underlying machine type. Even if compile-time verification is preferred to runtime verification whenever possible, many of these checks are still done dynamically. If they fail, an exception is raised at runtime. For example, calling the procedure `Increment` below on `Integer'Last` will result in an exception as the increment of `X` will overflow.

```
procedure Increment (X : in out Integer) is
begin
   X := X + 1;
end Increment;
```

The addition of contracts in Ada 2012 follows a similar trend. If they are enabled at compilation, the boolean expressions supplied as pre and postconditions of subprograms or as type invariants are evaluated at runtime, and an exception is raised if this evaluation returns False. As an example, consider the following annotation for the procedure `Increment`:

```
procedure Increment (X : in out Integer) with
   Pre  ⇒ X ≠ Integer'Last,
   Post ⇒ X > X'Old;
```

If `Increment` is called on `Integer'Last` and the precondition is enabled at compilation, an exception will be raised as its boolean expression evaluates to False. Similarly, if a bug is introduced in the implementation of `Increment` so that its parameter `X` is no longer increased by the call, an exception will be raised when checking the postcondition.

SPARK is a static analysis tool for Ada. It allows users to verify that the language-mandated checks and the user-written contracts in their program will never fail at runtime without running the program. It is open-source[1] and available through the Alire package manager. In this article, we give insights on how the tool works in practice and what are the most important challenges in its implementation as of today.

[1]https://github.com/AdaCore/spark2014

2 Formal Proof of Programs

SPARK verifies programs at the source code level on all possible inputs at once using *deductive verification*[1, 2]. Figure 1 schematizes the different steps of the verification process. First of all, the user is responsible for annotating their pro-gram with contracts. These contracts can express properties the user wants to verify on their code, but we will see later that some contracts can be necessary even to verify language mandated checks. Then, the tool transforms the annotated program into a set of logical formulas called *verification conditions*. There can be one verification condition or more for each property that is verified on the Ada program, be it a check or a user-written contract. Finally, the verification conditions are given to automated solvers. If all the formulas are verified, the program is correct.

Deductive verification is modular on a per subprogram basis. It is necessary for the verification process to be scalable in practice. Contracts are used to summarize what the guarantees are for each subprogram, both from the caller's and form the callee's point of view. When analysing the subprogram itself, SPARK verifies that, for all inputs that fulfill the precondition, the subprogram executes safely (there are no runtime errors) and the postcondition holds on subprogram exit. As an example, let us consider the procedure `Increment` presented before. To be able to verify its body, it is necessary for the user to supply a precondition preventing it from being called with `Integer'Last`. With the contract proposed before, the SPARK tool is able to verify both that no overflows can occur during the increment, and that the postcondition necessarily holds at the end of the call.

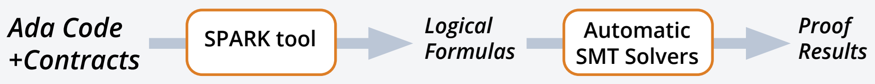

Figure 1: Deductive verification in SPARK

As the verification is modular, the SPARK tool does not look at the body of called subprograms, it only considers the contract: the precondition is checked and the postcondition is used to get information about the values of the objects modified by the subprogram after the call. As an example, let us consider the procedure Foo defined below. It calls `Increment` twice in a row on a variable initially initialized to 0:

```
procedure Foo is
   X : Integer := 0;
begin
   Increment (X);
   Increment (X);
end Foo;
```

For each call to `Increment`, the analysis tool needs to verify that the precondition holds. For the first one, the verification succeeds. Indeed, `X` is known to be 0 before the call, which is not `Integer'Last`. The verification fails however for the second call. Looking at the body of `Increment`, we know that `X` should be 1 at this point, so the precondition would evaluate to True if we were to execute `Foo`. The SPARK tool however, will fail to verify it. As it works modularly on a per-subprogram basis, it cannot look at the body of `Increment` when analysing `Foo`. Instead it looks at its postcondition, which is not precise enough to rule out the possibility of `X` being `Integer'Last` after the first call.

In general, to verify a program using SPARK, it is necessary to annotate all subprograms with contracts precise enough to entail together the correction of the complete program. The preconditions should be strong enough to verify the sub-programs themselves, and the postconditions to verify their callers. In our example, if we change the postcondition of `Increment` as below, then both procedures can be automatically verified:

```
procedure Increment (X : in out Integer) with
   Pre  ⇒ X ≠ Integer'Last,
   Post ⇒ X = X'Old + 1;
```

Unfortunately, the underlying logic used for deductive pro-gram verification is undecidable. As a result, it is not possible for such a tool to be able to decide on every program whether it is correct or not. Some tools, commonly called *bug finders*, focus on reducing the number of false alarms - they try to only report a bug when they are sure that there is one. SPARK on the other hand is a *sound* verification tool. If it can verify a program, then the program is correct[2]. However, it is not *complete* - if the verification fails, that does not necessarily means that the program is incorrect. It is also possible that some contracts are missing, or that they are not sufficiently precise to verify the program. As we have seen before, it is the case of the postcondition of `Increment` presented in Section 1 which is not strong enough to verify the procedure `Foo`. Finally, the background solvers might also be unable to verify a valid formula in the allocated time. This sometimes makes it difficult for users to understand why the verification of a program is failing. The SPARK tool tries to help the investigation by providing counterexamples whenever possible. Unfortunately, the incompleteness of of the background solvers means that this is not always possible, in particular when the program becomes more complex. Efficient and reliable counterexample generation for deductive verification is still being researched[3, 4].

3 Why3 as an Intermediate Language

The SPARK tool works by using in the background bleeding edge technology developed by academic researchers in the formal verification domain. The Why3 platform[5], developed at Inria in France, performs deductive verification on a ML like semi-executable language called WhyML. It generates the verification conditions and translates them into the input language of various automated or manual solvers. As schematized in Figure 2, Why3 is used as a backend by several tools targeted at mainstream programming languages such as C[6] or Java and more recently Rust[7]. SPARK is such a frontend for Ada.

Compared to other frontends of Why3, SPARK can take ad-vantage of the constraints imposed by the Ada language to reduce the need for user-written annotations and to make the verification process more efficient. In particular, the constraints imposed by the expressive type system of Ada are reflected on the Why3 side through assumptions. As an example, let us consider the record type `My_Rec` defined below. It has a discriminant `Length` which is used to bound the length of its array component F. The elements of the array are natural numbers bounded by a constant `Max`.

```
subtype My_Nat is Natural range 0 .. Max;
type Nat_Array is array (Positive range <>)
   of My_Nat;
type My_Rec (Length : Natural) is record
   F : Nat_Array (1 .. Length);
end record;
```

[2]The soundness of the tool relies on assumptions. For example, since the verification is done on the source code, it assumes that the program is compiled correctly.

In the generated WhyML code, values of type `My_Rec` are assumed to fulfil its *dynamic invariant:* a predicate which gives both the bounds of the array component `F` and the range of its elements. It is defined as follows:

```
predicate dynamic_invariant (x : my_rec) =
     first x.rec__f = 1
   ∧ last x.rec__f = x.rec__length
   ∧ ∀ i : int.
        first x.rec__f ≤ i ≤ last x.rec__f →
          0 ≤ get x.rec__f i ≤ max
```

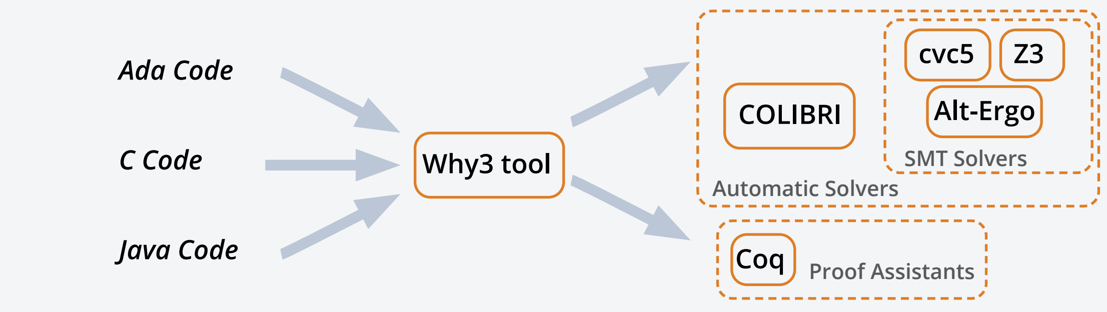

Figure 2: The Why3 platform for deductive verification

In general, the translation from Ada to WhyML does not at-tempt to preserve the executable semantics of the program, but rather to produce the simplest verification conditions possible while retaining soundness. In fact, the generated WhyML pro-gram is not even executable. For example, all subprograms are translated twice: once to generate the verification conditions for their body, and once to verify their callers. This has several advantages. In particular, it makes it possible to translate subprograms without worrying about the order of declarations and possible mutual recursivity. Indeed, fol-lowing the order in which the Ada subprograms are declared would not be enough to avoid forward references in WhyML, as Ada allows calling functions which have not been defined yet inside contracts. It also allows the SPARK tool to use different contracts for the subprogram depending on the calling context. This is important to handle dispatching calls on tagged primitives in particular.

As an example, here is the declaration of `Increment` used for the verification of `Foo`. It is an abstract function without a body. Its `writes` contract states that it modifies its input `x`. The `requires` and `ensures` annotations are direct transla-tions of the Ada contract. The fact that the parameter `x` fulfils the constraints of its Ada type after the call is made explicit in the postcondition.

```
val increment (x: ref int) : unit
   writes {x}
   requires {!x ≠ 2147483647}
   ensures
      { !x > old !x ^ dynamic_invariant !x }
```

When verifying increment itself, the translation is quite different. First of all, the parameters are declared as global variables instead of actual parameters of the subprogram. It makes it easier to verify entities nested inside the subprogram (a nested package or a nested subprogram for example). Then, while the Why3 translation of `Increment` has a postcondition, it does not have a precondition. This allows the SPARK tool to verify that the Ada precondition is *self-guarded*, that is, it can be evaluated in any context without raising an exception. Afterward, the Ada precondition is assumed before verifying the rest of the subprogram:

```
 (* The parameters of Increment are global variables *)
val x : ref int;
let increment__def (_ : unit)
   ensures { !x.int__content > old !x }
=
   (* Assume that x follows the Ada typing
      rules *)
   assume { dynamic_invariant ! x };
   (* Check that no runtime error can occur
      while evaluating the precondition *)
   (begin
      ensures { true }
      let _ = !x ≠ 2147483647 in ()
   end);
   (* Assume the precondition of Increment *)
   assume { !x ≠ 2147483647 };
   ...
```

4 Different Solvers in the Background

To discharge the verification conditions, SPARK relies mostly on *Satisfiability Modulo Theory (SMT)* solvers. These auto-mated solvers are well-suited for program verification because they support natively theory symbols, that is, symbols which have a generally understood meaning outside of the context of the verification condition. These symbols include in particular integer or floating point literals and arithmetic operators, which occur often in programs. In general, a verification condition coming from an Ada program references symbols from several theories. As an example, the dynamic invariant of type `My_Rec` defined in Section 3 uses symbols from:

- ***the theory of linear arithmetic on mathematical integer types for the integer literals and the comparison operators,***
- ***the theory of abstract data-types for record components, and***
- ***the theory of infinite immutable arrays for the access to the array component.***

It also uses the universal logical equality symbol, first-order quantification, and uninterpreted function symbols like `max`.

As stated before, the resulting logic is undecidable. It means that it is not possible to design an algorithm which would be able to determine on all such logical formulas whether they are valid or not without errors. The solvers used in the backend of SPARK are all sound - they never verify an invalid formula. However they are incomplete, so they might not be able to verify valid formulas. In particular, certain constructs are not efficiently supported by any solvers and are the subject of active research. Here are some of these topics:

- ***first-order quantification, in particular with alternating quantifiers,***
- ***non-linear integer arithmetic, which is undecidable even on quantifier free formulas, and***
- ***conditions involving symbols from different theories -floating point numbers and integers for example.***

To alleviate these concerns, SPARK takes advantage of the capability of the Why3 platform to target different provers. By default, it uses three SMT solvers as a backend, Alt-Ergo[8], cvc5[9], and Z3[10].

All are independent open-source tools developed, at least initially, as part of a research endeavor. The three solvers are run on every verification condition, one after the other, until the condition is proved. This allows users to take advantage of the different strengths of these solvers.

In addition, as Why3 makes it possible to tune the translation specifically for each solver, the SPARK tool increases the diversity between the solvers by purposefully encoding the problem differently for each of them. Indeed, the best way to encode a feature of the language might depend on the use-case, and having several increases the chance of finding a proof. As an example, consider modular integer types. They can be encoded either a a mathematical integer, or as a bitvector of fixed size. The first encoding is often preferable when doing standard integer manipulation as well as when converting toward other numeric types such as signed integers and floating-point numbers. The second makes the support of bitwise manipulation more effective. To get the advantages of both world, the SPARK tool uses bitvectors for cvc5 and Z3, and mathematical integers for Alt-Ergo.

Even if SMT solvers are better suited for program verification, the use of other kinds of solvers, which might not suffer from the same caveats, is also investigated. In particular, constraint solvers could improve the provability on quantifier-free verification conditions involving conversions between values of different theories. The solver COLIBRI[11] is already available from the SPARK verification tool, though it is not used by default yet.

In general, choosing which provers to run and with which options and encoding is a question of trade-off. Running one more prover is time consuming, as it will be launched on all the verification conditions, and possibly run up-to the provided time-limit. This is why it is important to limit the number of solvers used by the tool, and to assess the efficiency and interest of each new addition thoroughly.

5 Expressivity versus Efficiency

Taking a step back, we have seen that verifying Ada programs using deductive verification is complex, and requires user input, in particular in the form of contracts. To make the tool usable in practice, it is important to find a balance in the accepted language between expressivity and ease of annotation and verification. For that, the SPARK tool introduces simplifying assumptions. At the simplest, these assumptions are features of Ada which are not supported by the tool, like side-effects in functions, or handling of exceptions. Others are more complex, for example all values are assumed to be valid, and no two objects can be aliases of each others -modifying an object cannot affect another object in a visible way.

For the SPARK tool to remain sound, it is important that it verifies that these assumptions are valid on the program. This is generally ensured by a separate analysis done by the SPARK tool, prior to running deductive verification. As part of this effort, a program flow analysis is run on the code to determine the effect of all subprograms and the global data that they access. Based on the results of this analysis, it is possible for the SPARK tool to ensure that functions do not have side-effects, and that no uninitialized variables can be read, which helps to rule out invalid values. Absence of aliases in programs using pointers is enforced through an ownership policy with its own specialized analysis.

Simplifying assumptions need to be chosen carefully so they are both effective - they reduce significantly the complexity of either the verification itself or the manual annotation process - and not overly restrictive. The right balance is generally hard to find, and can be refined in subsequent releases of the tool. For example, by default, SPARK enforces correct initialization of variables by a strict initialization policy: all inputs of a subprogram shall be entirely initialized at the point of call and all its outputs shall be entirely initialized when the subprogram returns. This restriction is useful, as it saves the user from having to annotate all their subprograms with contracts about initialization of values. However, it makes it impossible to annotate and verify a program which initializes a record component by component in separate procedures as the program over the page:

```
type Two_Fields is record
   F, G : Integer;
end record;

procedure Init_F (X : in out Two_Fields) is
begin
   X.F := 0;
end Init_F;

procedure Init_G (X : in out Two_Fields) is
begin
   X.G := 0;
end Init_G;

procedure Process (X : in out Two_Fields) is
   ...

procedure Main is
   X : Two_Fields;
begin
   Init_F (X);
   Init_G (X);
   Process (X)
end Main;
```

Since the parameter X of `Init_F` has mode in out, it is an input of the subprogram and SPARK will try to verify that it is entirely initialized before the call to Init_F in Main. Obviously, it is not the case. Unfortunately, changing the mode of the parameter X of Init_F to out will not solve the issue as the tool will then try to verify that X is entirely initialized by Init_F, which is not the case either. To alleviate this issue, more recent versions of SPARK allow users to annotate their objects to exempt them from the initialization policy. However, it then becomes necessary for the user to manually add information about initialization in subprogram contracts:

```
type Two_Fields is record
   F, G : Integer;
end record;

procedure Init_F (X : in out Two_Fields) with
   Relaxed_Initialization ⇒ X,
   -- X is no longer subjected to the
   -- initialization policy of SPARK.
   Post ⇒ X.F'Initialized
   -- X.F is initialized by Init_F
   and (X.G'Initialized = X.G'Initialized'Old)
   -- X.G is left as it was
is
begin
   X.F := 0;
end Init_F;

procedure Init_G (X : in out Two_Fields) with
   Relaxed_Initialization ⇒ X,
   Post ⇒ X.G'Initialized
   and (X.F'Initialized = X.F'Initialized'Old)
is
begin
   X.G := 0;
end Init_G;
```

```
procedure Process (X : in out Two_Fields) is
   ...
-- The parameter X is not exempted from
-- initialization checks.

procedure Main is
   X : Two_Fields with Relaxed_Initialization;
begin
   Init_F (X);
   Init_G (X);
   -- Complete initialization of X is
   -- checked at this point.
   Process (X);
end Main;
```

The parameter `X` of `Init_F` is exempted from the initialization policy, so SPARK no longer tries to verify that it is initialized before the call in `Main`. Without additional an-notations, it does not assume that it is initialized afterward either. It makes it necessary for the user to add a postcondition to `Init_F` to say that it initializes the component F of its parameter. The postcondition also says that the component `G` is not uninitialized by the call. It is not necessary to verify `Main`, as `Init_F` is called before `Init_G`, but it preserves the symmetry between `Init_F` and `Init_G`. The parameter `X` of Process does not need to be exempted from the initialization policy, as Process is necessarily called on entirely initialized data. As a result, we do not need to supply a contract about initialization for it. With these annotations, the code is entirely verified by the SPARK tool.

All these restrictions imposed by SPARK on top of Ada define a language subset. The language supported as input of the SPARK tool is called the *SPARK language*. It is not a strict subset of Ada, as can be seen on Figure 3, as it also introduces additional annotation features. As an example, *contract cases* are used to specify a contract as several smaller contracts with their own precondition and postcondition, and code which is only used for the specification process can marked as ghost so that it is removed by the compiler when the assertions are not enabled at runtime. This language evolves continuously as more features are added to the tool. Major additions to the SPARK language are discussed online[3] and community input is always valuable.

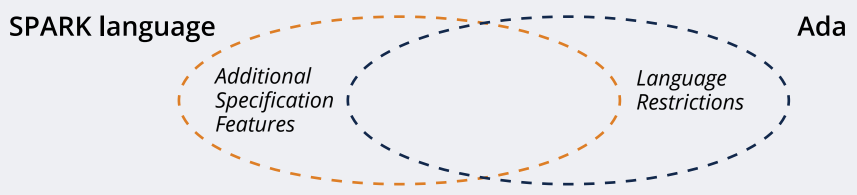

Figure 3: The SPARK language

6 Conclusion

SPARK is a static analysis tool for Ada. It verifies that no language mandated checks can fail at runtime, and that user written contracts always hold. The analysis is modular on a per subprogram basis: when analyzing a subprogram, the tool only uses the contract of called subprograms and not their bodies. As a consequence, the tool requires users to annotate all their subprograms with contracts precise enough to ensure together the correction of the whole program.

The SPARK tool is based on the Why3 plateform for program verification. The program and the contracts are transformed into a set of logical formulas which are then verified by auto-matic solvers. The solvers used as the backend of SPARK are mostly SMT solvers. They work on first-order formulas with interpreted symbols coming from various theories - integer and floating-point arithmetic,

[3]https://github.com/AdaCore/ada-spark-rfcs

bitvectors… This logic is undecidable. To work around prover's limitations, SPARK uses several automatic solvers for the verification.

To make both the verification and the manual annotation process tractable, SPARK introduces simplifying assumptions. These assumptions are associated to language restrictions which are verified by the tool. Both the SPARK proof tool and the related language restrictions are evolving continuously. For example, support for access types and its ownership policy have been added and extended in the last couple of years. Precise support for handling of exceptions is being discussed currently.

References

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[2] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[3] S. Dailler, D. Hauzar, C. Marché, and Y. Moy, "Instrumenting a weakest precondition calculus for counterexample generation," *Journal of logical and algebraic methods in programming*, vol. 99, pp. 97–113, 2018.

[4] B. Becker, C. Lourenço, and C. Marché, "Explaining counterexamples with giant-step assertion checking," in F-IDE 2021-*6th Workshop on Formal Integrated Development Environments*, Electronic Proceedings in Theo-retical Computer Science, 2021.

[5] J.-C. Filliâtre and A. Paskevich, "Why3—where programs meet provers," in *European symposium on programming*, pp. 125–128, Springer, 2013.

[6] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *International conference on software engineering and formal methods*, pp. 233–247, Springer, 2012.

[7] X. Denis, J.-H. Jourdan, and C. Marché, *The Creusot Environment for the Deductive Verification of Rust Programs*. PhD thesis, Inria Saclay-Île de France, 2021.

[8] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout, "Alt-ergo 2.2," in SMT Workshop: *International Workshop on Satisfiability Modulo Theories*, 2018.

[9] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, et al., "cvc5: a versatile and industrial-strength smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 415–442, Springer, 2022.

[10] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[11] B. Blanc, C. Junke, B. Marre, P. Le Gall, and O. Andrieu, "Handling state-machines specifications with gatel," *Electronic Notes in Theoretical Computer Science*, vol. 264, no. 3, pp. 3–17, 2010.